ALEXANDER GALLOWAY 2016-01-12

THE COMPUTATIONAL DECISION

GENERICSCIENCE MEDIA THEORY, TECHNOLOGY

As various forms of computationalism continue to influence contemporary writing in theory and philosophy, it seems appropriate to reexamine some of the fundamental assumptions and principles of computation, so that we may identify them more readily. In an earlier, characteristically modernist, phase, media theorists like Lev Manovich and others sought to identify and itemize the unique characteristics of the medium by pursuing the query “what is new media?”

I’ll add my own contribution here, by returning to some of the basic observations about data and machines. Still, the label “computational decision” is meant to indicate several positive actions undertaken in order to create and sustain computation in the world. Such actions are not always taken consciously, nevertheless they are taken and must be taken for computation to exist. The notion of a “decision” is meant to highlight the constructed nature of such events. Further, I suspect that anyone promulgating such decisions might also wish to provide sufficient rationale for their naturalization, least risk being accused of dogmatism.

The three basic moments in the computational decision are:

“there are data,” or the decision to pursue *symbolic representation*

“there is information,” or the decision to grant a macro *structure* to data

“there are functions,” or the decision to distinguish between two kinds of information, *functions and (mere) data Data*,

information, and function. These three moments are cumulative, with one step building on the previous step. And while specific technologies intervene at each moment — with things like scanners or cameras residing at the first moment, protocols at the second moment, and subroutines or black boxes at the third moment — the computational decision is, ultimately, an integrated process in which all three moments come together to work in concert. Thus, all three moments can be combined into a single, albeit verbose, definition: The computational decision is the decision to structure symbolic representation in terms of function and data.

Before explaining these three moment more fully, let me offer a few caveats in the hopes of fending off some misunderstandings.

First, while there are many different kinds of computers and the history of computation is filled with curious experiments and interesting exceptions, I take it as given that computers are all essentially *imperative*. All computers execute imperative statements in order to change memory states — at least I take that as true for the following discussion. Second, I don't consider the various programming paradigms such as functional or object-oriented to be so dramatically different from one another as to form hard boundaries. Different languages may have different scales of abstraction, but most languages use the same basic kinds of abstraction. In other words, a struct and an object may be technically different, but they are, in my view, not philosophically different. An object is a set of values stored in memory, just like a struct, or a primitive data type. There is not much that can be said about an object method that can't also be said about a subroutine, or for that matter any of the basic operations like addition or subtraction. Finally, as a third caveat, I don't consider hardware and software to be fundamentally different. All hardware relies on various forms of embedded abstraction and formalization, just as there is no software — except in the impossible recesses of imagination — that does not operate in a material milieu. Please consider these caveats before hailing me with protestations of the form "but Lisp works in such and such a way…"

About the first moment ("there are data") I will say relatively little, since I have already discussed it briefly before on this blog, and because, to be frank, it pertains to wildly speculative and contingent historical behavior. The phenomenological notion of data as "the givens" approaches those far limits of comprehension that are best taken on faith. Or, as in the rather contrary instance of scientific positivism, the decision to measure and interact with the world via quantified symbolic representation should only be studied within the context of specific socio-cultural evolution, entailing technical and industrial histories, studies over labor and class struggle, the promulgation of Enlightenment rationality around the world, and other inquiries of a more sociological or ethnographic nature.

So we shall vouchsafe that mysterious domain of givenness to phenomenology for the time being, or wherever else it may lie, and proceed to the question of information. Information is the marriage of data and structure. Information means a series of letters or numbers (or any kind of fixed inscription of a legible substrate) organized according to an abstract formal structure. The structure could be as simple as saying "assume that high voltage means 'yes' and low voltage means 'no'" or "assume that eight bits in a row constitute a single number." In each case a layer of abstraction is added to a series of legible given realities.

Such formal abstractions build on each other in a cumulative way. Thus one could also assume that a series of bytes make up an integer. Or, at a higher level of abstraction, one could assume that an object contains a velocity and a direction. Each level of abstraction builds on a lower level. Each level consists of numbers organized according to a given formal structure.

(This leads to some interesting scenarios, because, as hackers and artists have shown, it's trivial to strip data of its formal layer and superimpose another abstraction in its place. The data doesn't know the difference. The phenomena of datamoshing illustrates this quite vividly, producing its effects by pairing encoded digital video data with incorrect instructions for its decompression.)

The second moment ("there is information") thus leads directly to the third because, within the structuration of data, one particular structure is more important than all the rest. This is the decision to mark a distinction between mere data and the various operations that take place on that data. The third moment thus states that "there are functions" over and above mere data. Here I am merely reiterating a basic truth of computer science, as described in books like Niklaus Wirth's revealingly titled *Algorithms + Data Structures = Programs* (1976). I gravitate to the term function because of its interesting history and relevance in a number of fields including mathematics and computer science. But other terms should also be included here, depending on context, terms like algorithm, operation, procedure, and machine. (Recall how, in Deleuzian theory, machines resemble functions in the sense that they operate on materials and produce a transformation of those materials — the heart machine, the war machine, and so on.)

Interesting parallels can be made here with Badiou's concept of "democratic materialism," or the dogmatic position that, as he puts it, "there are only bodies and languages." Today's computational materialism says something similar, that there are only data and operations. Data take the place of bodies, while operations takes the place of languages. Everything else stems from this. The fundamental decision is made formally — in fact, as the second moment indicates, it *must* be made formally because, as sequences of numbers, there is no essential distinction between data and operations. Bytes are data. Variables are data. Structs and tuples are data. But addition and subtraction are also operations on data. And functions and methods are larger enclosures containing sequences of operations. (We might return, then, to Wendy Chun's work on storage, given that the basically agnostic nature of computational storage media, be it volatile memory or more long term storage, forces us to superimpose artificial form on data; there's no "algorithmic storage" that is qualitatively different from "data storage" in the way that, as certain philosophers would maintain, mind is different from body.)

But what is a function? A function is a device for correlating one set of things to another set of things. In the context of mathematics, a function associates one set of numbers with another set of numbers. The computational definition is similar: a function (a subroutine, a method, etc.) accepts certain inputs and produces certain outputs (where, incidentally, the existence of both input and output are optional, since there are some functions that do not require inputs and some functions that return no value).

The function has a rather interesting architecture, and I'll elaborate two main qualities of this architecture that I find most interesting. The first we'll call "relational mapping" or "input-output mapping," and the second is called "encapsulation."

First, "relational mapping" means that functions map a set of inputs to a set of outputs. This can be expressed formally as $f(x) = y$, where f represents an operation that maps x to y. For example, if x is 3 and f is the operation "add two" then y is 5.

input is 3 → add two → output is 5

Yet functions can operate on variables, not just individual numbers. This means that they operate on *sets* of values within a certain type. Thus if x is the set "2, 4, 5" and f is the operation "add two" then y is the set "4, 6, 7."

"2,4,5" → add two → "4,6,7"

This is the basic structure of procedures and functions in computing. However most computer languages don't look like this (although some get pretty close). A more common syntax for declaring functions in computer languages is as follows:

output operationName(input) { operations }

Hence we can rewrite the the starting example as follows:

"4,6,7" functionForAddingTwo("2,4,5") { add two }

Note it's often best *not* to read left-to-right when parsing these kinds of declarations. One must simply identify the components and become familiar with their arrangement. Nevertheless, when read left-to-right, this states that "an output, is returned from a function, which accepts an input, and the function is defined in such and such a way."

Finally, the trailing reference to operations is often suppressed, resulting in a shorter format

"4,6,7" functionForAddingTwo("2,4,5")

which, if we replace "4, 6, 7" with the variable y and "2, 4, 5" with the variable x, brings us back to where we started: $f(x) = y$.

Swift, which I've been exploring recently, has a slightly different syntax, changing the order of some of the components. Still, the semantics are the same even if the syntax is different:

func operationName(input) -> output { operations }

What this says in longhand is that "operationName" uses "operations" to map an "input" value to an "output."

This is the basic structure of what we have labeled "relational mapping" or "input-output mapping." One set of values is mapped over to another set of values. Given certain input, a function will produce a certain output.

An important element was suppressed along the way, however, the element within the curly brackets labeled "operations." This is a crucial detail and it's worth exploring a bit more. Simply by defining the function as a relational mapping, while hiding the operations, it is not shown *how* such relational mapping occurs. Inputs arrive and outputs depart, as if by magic. Such apparent neglect of "the how" is, in fact, a conscious design feature. Functions are explicitly designed in such a way as to uncouple knowledge about the relational map (*that* set *A* relates to set *B*) from knowledge about the specific mechanics of transforming between sets (*how* any member of *A* transforms into a member of *B*). The name for this is "encapsulation."

Encapsulation refers to the separation between declaration and implementation. It means that functions can and should be defined in simple, abstract terms, separate from the low-level details of how they are implemented. The distinction is illustrated above by the separation made between the declaration in the form $f(x) = y$, and the implementation contained within the curly brackets. Hence $f(x) = y$ is simply shorthand for $f(x)$ { operations } = y in which "{ operations }" refers to the things that must be done to map x onto y. Such operations are "encapsulated" or isolated into their own machinic and semiotic space.

The reason why this is important is that a partition can be erected between the declaration of the function and the implementation of the function. Again, to be clear, the partition would exist here between "$f(x) = y$" on the one hand and "{ operations }" on the other. The operations are encapsulated into their own module and set aside. In fact many languages make this partition very explicit, putting the declaration of the function in one file, and the implementation of the function in another file altogether.

Let me stress that *all* operations can be considered using the paradigm of the function whether or not they adopt that name. Hence, low level operations like addition and subtraction follow this paradigm, but so do mid level constructs like methods and functions, as well as larger structures like interfaces and APIs. In other words there's nothing formally different between adding two numbers and querying a Web service. In both cases, an input is supplied to an operation, which returns an output.

In popular parlance, encapsulation is often called *black boxing*. Black boxing has its origins in World War II and early cybernetics research, but it essentially refers to a scenario in which the innards of a function (or some other mechanical device) are obscured to outside observers, while the inputs and outputs remain legible. Hence the following is a black box:

"4,6,7" function("2,4,5") { ??? }

In fact, *all* function declarations of the form "*f*(*x*) = *y*" are black boxes. Black boxing simply means that the outward "signature" of the function is legible, while the inner mechanism is illegible.

Boxes beg to be opened, of course, and human beings are not known for curbing their own curiosity. Thus the fundamental question of the black box is how to determine the innards of the box without being able to open it. Or, using the above example, how to interpolate the "???" from only having access to "2,4,5" and "4,6,7."

Entire sciences were invented to solve this problem. And, indeed, once solved, subsidiary sciences were subsequently invented to profit from the logic of the black box. Given a certain amount of regularity between input and output, the obscured operations can be guessed reasonably well. Statistical modeling helps solve some of the more unpredictable black box problems. And, in other cases, the logic of the black box was so prevalent that deciphering it became less important. Game theory and behaviorism, for instance, are premised on the assumption that opening the box is ultimately unnecessary. The black box has also been important in philosophy of mind, as with John Searle's so-called Chinese Room problem (which he used to show that computers could appear to think without actually thinking). It's difficult to understate the importance of this paradigm within twentieth-century science and humanities. In general, black boxing indicates an epistemological shift from knowing-how to knowing-that. And, at the same time, it indicates a willful obfuscation of one segment of the function (knowing-how) in favor of the full and clear revelation of the other.

Note that this isn't a conspiracy theory; it's often possible to open the black box in order to inspect its implementation. However this is not always necessary and, in many cases, not advantageous from the perspective of usability and efficiency. I usually don't need to know *how* my CPU adds floating point numbers; I only need to know *that* it does. And if, eventually, I wish to learn how my CPU performs such miracles, I can do so provided I'm willing to invest the time and energy required to open the black box (that is, to learn a little bit about electrical engineering and master the particular op codes and machine language used in my CPU). This is how Bruno Latour uses the concept of the black box in his work. For him, black boxes are economic conduits for the mutual exchange of practical action and technical knowledge. Any black box may be opened, provided one is willing to pay the cost.

Given the separation between declaration and implementation, some sort of interaction is required to reconnect them. Here and elsewhere computation reveals its stunningly metaphysical design. Encapsulation thus necessitates something else that is very important in computation, *an interface*. An interface is, quite literally, nothing more than the relationship between a function's declaration and its implementation, or between "*f*(*x*) = *y*" and "{ operations }." In fact, a function's declaration is often simply understood as the function's interface. Larger data structures like objects will have more complex interfaces, consisting of aggregations of things. And all other kinds of interfaces, from web pages to ATM screens, are merely larger versions of this basic scheme.

As I already hinted, there is a long complicated history for how computer languages implement the concept of an interface. For example, languages like C++ split the implementation of an object into two separate declarations, which are conventionally housed in two separate files. Languages like Java forgo separate declarations. But don't let that fool you: Java classes expose an interface signature just like C++ classes do; a Java class announces public member variables along with the signatures of its public methods. (Java also makes this explicit with the category of the "interface" itself, a separate declaration of methods that a class can choose to implement. In Swift these are called "protocols.")

When they are used in a library or a program, aggregations of interfaces are known as an API (Application Programming Interface). But these examples are all formally similar: a function signature, an object interface, and an API all follow the design pattern in which legible interfaces are linked to illegible innards.

Seen in this way, interfaces are a direct consequence of relational mapping, and, to the extent that functions require relational mapping, are thus a direct consequence of the computational decision as a whole. (Functions could, quite simply, be renamed "relational interfaces" without any change in meaning.) In other words, if we wanted to identify all the interfaces that surround us, we would have to identify the basic technologies of mathematics and computation, not simply our televisions and touch screens.

I will say relatively less about interfaces here, except to highlight how interfaces are essentially technologies of abstraction and formalization in which the parameters of type and structure are strictly defined. The interface of a bank ATM and the interface of a mathematical function are, in this way, formally identical. They both reveal a highly codified, abstract mechanism for accepting a particular kind of input. They both furnish a particular kind of output. And they both reveal very little about how the input was turned into the output (only *that* it was).

And, interestingly, some programers also promote an "interface-oriented" coding style, in which the programmer is meant to focus almost exclusively on authoring interface declarations, with the assumption that the implementation will be filled in later, likely by other interface declarations! This nested structure, in which the "content" of an interface is simply another interface, is but another consequence of the functional mandate of the computational decision. And it demonstrates the fundamentally mereological nature of computation, an "orthogonal" structure tracking up and down between systems and subsystems. There are, as it were, interfaces all the way down.

These are not the only things worth talking about, of course, and a longer discussion of the computational decision would doubtless address the question of modularity (stemming from the above interface architecture), or the question of hierarchy and inheritance, which is particularly salient, but not exclusively so, for object-oriented computer languages. Ian Bogost's book *Unit Operations* is a good place to start for such themes.

*What does this all mean? It means, in part, that we ought to have a healthy amount of skepticism toward those who flirt with various forms of computationalism — I adopt the term from David Golumbia's usage — in contemporary theory and philosophy, particularly those who make the concept of "machine" or "function" central to their work. This is not to say that such terms should be prohibited. On the contrary, my hope is to see more exploration of such terms, not less. Still, the naturalization of such concepts, or the mere assertion of their existence, if not universal dominance — "everything is information" or "everything is a machine" or "everything is a function" — should not be taken as gospel truth. Instead we ought to recontextualize the various components of the computational decision within the scientific and industrial milieu from which they emerged.

The question "what is new media?" is still an interesting one and worth pursuing. Yet a subtle shift might be in order today, from that question to a different one in which computation is understood as a decision within thinking.

taken from here

foto by Bernhard Weber

← PREVIOUS NEXT →
